Relazione Progetto:

Il Gioco della Vita

Cluster di RaspBerryPi

Realizzato da:

Davide Impiombato Mat: 166750
Salvatore Spagnuolo Mat: 182801

Il Gioco della vita:

L'automa cellulare che abbiamo realizzato è la versione standard del gioco della vita realizzato seguendo le seguenti regole di vicinato:

- Qualsiasi cellula viva con meno di due cellule vive adiacenti muore, come per effetto d'isolamento
- Qualsiasi cellula viva con due o tre cellule vive adiacenti sopravvive alla generazione successiva
- Qualsiasi cellula viva con più di tre cellule vive adiacenti muore, come per effetto di sovrappopolazione
- Qualsiasi cellula morta con esattamente tre cellule vive adiacenti diventa una cellula viva, come per effetto di riproduzione

(il vicinato è calcolato sulle otto posizioni adiacenti alla cellula)

Il Codice:

Per lo sviluppo dell'automa sono state utilizzate le seguenti librerie:

- MPICH 3
- Allegro 5

Il programma è stato scritto ad oggetti utilizzando il linguaggio C++ standard 11 e compilato con le librerie allegro, sono stati creati solo due oggetti:

- Master.h
- Slaves.h

Ed un Main:

- Main.cpp

In seguito, compilati con il compilatore per C++ di mpich più il link alle librerie di allegro:

```
$ mpic++ <Nome_Codice.cpp> -o <Nome_Output.out>  -lallegro
-lallegro_primitives
```

Il Comportamento del Main:

Il Main si preoccupa di generare le corrette istanze per ogni processo, darà di default al processo "0" l'istanza del master e a tutti gli altri quella degli slave.

```
//  Creo l'istanza del master
    if (id == 0) {
        start_time = MPI_Wtime();
        master_ptr = new master(SIZE, slaves_number, &status, &segment_snd, &segment_rcv, benchmark, bench_loop);
        if (benchmark) master_ptr->seed_cells_generator(24241, ((SIZE * SIZE) * 0.4));
        else master_ptr->random_cells_generator((SIZE * SIZE) * 0.4);
    }
//  Creo l'istanza degli schiavi
    else {
        slave_ptr = new slaves(SIZE, slaves_number, &status, &segment_snd, &segment_rcv);
    }
```

Subito dopo dopo inizia il vero core del programma. In un loop controllato dal master (grazie al ritorno della condizione booleana sulla variabile "finish" che poi viene rispedita in Broadcast a tutti i processi) vengono eseguiti tutti i metodi necessari al funzionamento dell'automa, tutte le comunicazioni sono bloccanti quindi tutti i processi si sincronizzano da soli fra loro.

```
//  Inizia il ciclo delle generazioni dell'automa cellulare
    while(!finish){
        if(id == 0) {
            master_ptr->send_matrix_to_salves();
            master_ptr->recive_matrix_from_slaves();
            if (benchmark) finish = master_ptr->run_benchmark();
            else finish = master_ptr->run_allegro();
        }
        else {
            slave_ptr->recive_from_master();
            slave_ptr->compute_matrix();
            slave_ptr->send_to_master();
        }
//  Comunico a tutti i processi lo stato della variabile finish
        MPI_Bcast(&finish, 1, MPI_CXX_BOOL, 0, MPI_COMM_WORLD);
    }
```

Infine, fa stampare al master tutte le statistiche dell'esecuzione e libera la memoria dei vari processi.

```
//  Stampa le statistiche della sessione e libera la memoria
    if (id == 0) {
        finish_time = MPI_Wtime();
        std::cout   << "Generazione: " << master_ptr->getGenerazione()  << std::endl
                    << "Cellule ancora vive: " << master_ptr->cells_alive() << std::endl
                    << "Tempo di esecuzione in secondi: " << (finish_time - start_time) << std::endl;
        delete master_ptr;
    }
    else delete slave_ptr;
//  Finalizzo MPI
    MPI_Finalize();
    return 0;
} // Main
```

Il Master:

Contiene l'intera matrice M [N + 2] [N] (le due righe in più che fungono da bordo superiore e inferiore, valgono sempre zero). Si preoccupa di suddividere la matrice equamente, invia ad ogni schiavo la propria porzione e anche i bordi della sezione precedente e successiva. In seguito si mette in attesa della ricezione delle porzioni rielaborate dagli schiavi per ricomporre totalmente la matrice e generare un l'output grafico.
Di seguito l'implementazione del "Send" che invia le varie porzioni a tutti gli schiavi e la "Recive" che ri-assembla la matrice rielaborata e inviata dagli schiavi.

```
void master::send_matrix_to_salves() {
    int index = 0;
    for (int i = 1; i <= n_slaves; i++) {
        MPI_Send(&matrix[index][0], 1, *segment_snd, i, 0, MPI_COMM_WORLD);
        index += row_for_slaves;
    }
}

void master::recive_matrix_from_slaves() {
    int index = 1;
    for (int i = 1; i <= n_slaves; i++) {
        MPI_Recv(&matrix[index][0], SIZE * row_for_slaves, MPI_CXX_BOOL, i, 0, MPI_COMM_WORLD, status);
        index += row_for_slaves;
    }
    count_loop++;
}
```

Gli Schiavi:

Al loro interno hanno una matrice parziale M_Read[(N / n_slvaves) + 2][N] per la lettura, ed un'altra M_Write[(N / n_slaves)][N] per la scrittura del risultato dell'elaborazione. M_Read contiene due righe in più perché riceve dal master anche i rispettivi bordi delle sezioni di matrice precedente e successiva rispetto alla propria porzione.
Gli schiavi si preoccupano di ricevere la loro porzione su M_Read, elaborare e scrivere i risultati su M_Write. In seguito, l'intera M_Write sarà rispedita al Master che si preoccuperà di ricomporre la matrice. L'implementazione delle loro "Send" e "Recive" è la seguente:

```
void slaves::recive_from_master() {
    MPI_Recv(matrix, SIZE * (row + 2), MPI_CXX_BOOL, 0, 0, MPI_COMM_WORLD, status);
}

void slaves::send_to_master() {
    MPI_Send(matrix_tmp, 1, *segment_rcv, 0, 0, MPI_COMM_WORLD);
}
```

L'architettura del cluster:

Per il progetto abbiamo deciso di dividere il calcolo in MPI del gioco su di una matrice M [70] [70] su otto processi distribuiti su due nodi (RaspBerryPi 3B).

Il primo nodo contiene il processo master e tre slave, il secondo i restanti quattro slave. I nodi sono stati collegati localmente tramite cavo CAT45 (ethernet) diretto, il node1 è anche collegato ad un router Wi-Fi dove è connesso anche un PC, in questo modo il PC si collega al node1 tramite un SSH grafico X11 per visualizzare a schermo l'output del node1.

```
$ ssh -X pi@<ip address node1>
```

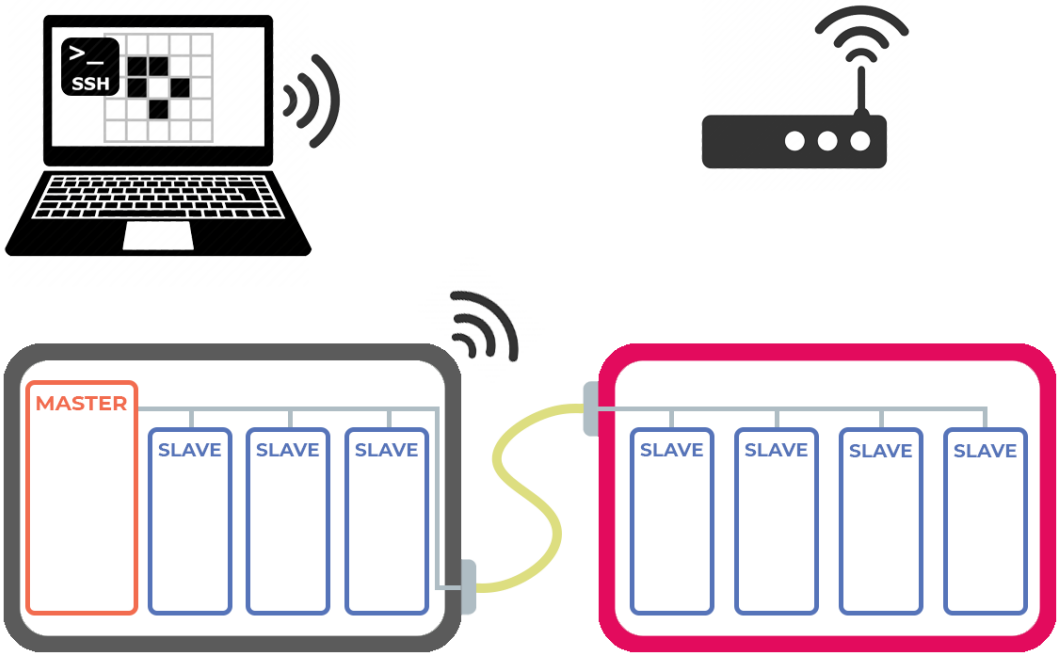
Per poi eseguire l'automa con:

```
$ mpiexec -f <MachineFile> ./<Eseguibile>
```

Nota: il comando deve essere eseguito dall'interno di una cartella Cloud condivisa tra i due nodi dove all'interno sono posizionati sia l'eseguibile che il machinefile.

Quindi il programma verrà eseguito con il relativo output grafico di Allegro.

Schema dell'architettura del cluster:



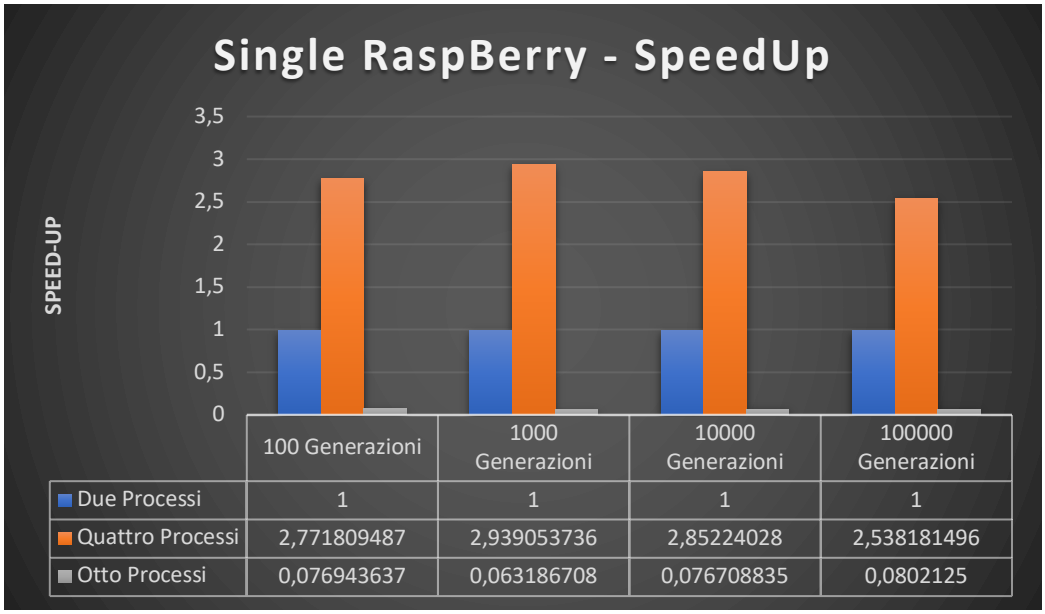
Le Principali caratteristiche:

- L'automa si basa su un'architettura **Master-Slave**.
- Sono state utilizzate delle matrici statiche per consentire l'utilizzo dei **DataType** di MPI.
- Si può eseguire dinamicamente su 'n' schiavi senza ricompilare il codice, a patto che 'n' sia divisibile per il numero di righe della matrice e che ogni schiavo abbia una matrice sufficientemente grande per ospitare la propria sezione.
- Anche la dimensione della matrice può essere cambiata molto facilmente, basta modificare la variabile SIZE nel Main e la grandezza effettiva delle matrici nelle classi stesse (in questo caso necessita una ricompilazione).
- Le matrici sono pensate per essere N x N, non sono ammesse altre configurazioni.
- È stata implementata una modalità "**benchmark**" la quale genera lo stesso pattern di cellule ad ogni esecuzione ed alla fine di 'n' generazioni definite dall'utente, ritorna le tempistiche di esecuzione e il numero di cellule ancora vive. Quest'ultimo serve per determinare in maniera approssimativa la corretta esecuzione del programma, in quanto il numero dovrebbe risultare identico su distinte esecuzioni (su 'p' Processi) al pari di "SIZE" di M e di 'n' generazioni.
- Per ottimizzare la memoria utilizzata da ogni processo, le istanze vengo chiamate sulla memoria dinamica, cosicché ogni processo ha un **puntatore** di tipo "Master" ed uno di tipo "Slaves" ma solo su uno di essi viene chiamata un'istanza "**new <Master/Slaves>**". In questo modo, la memora statica di un Master, non verrà mai neanche allocata ad uno Slave e viceversa.

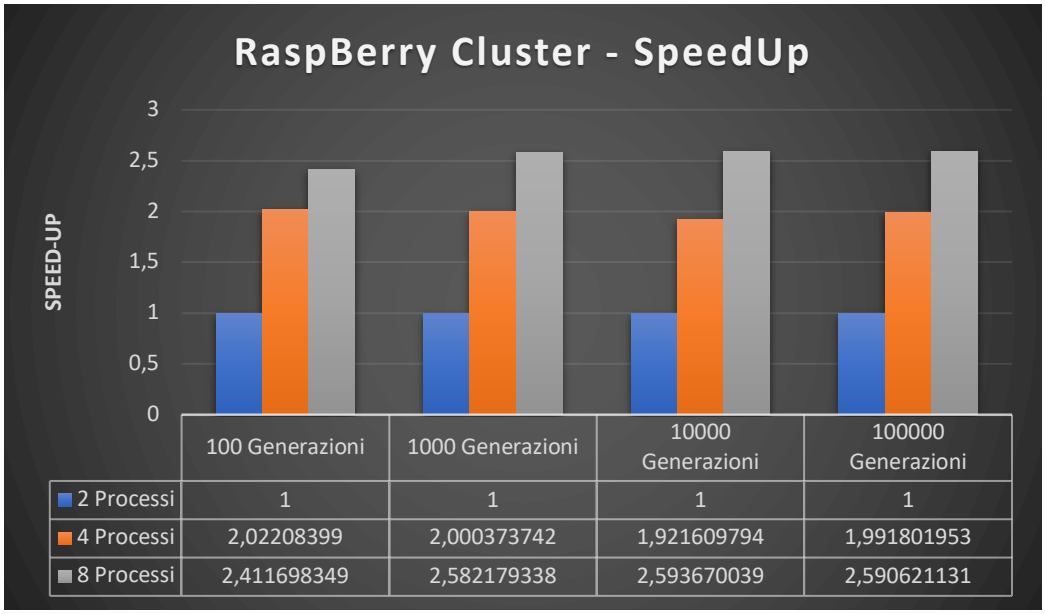
Informazioni utili sulle misurazioni:

- Tutti i **Benchmark** sono stati eseguiti su una matrice **M [105] [105]**.
- Gli **Speed-Up** sono stati calcolati prendendo come base i tempi del benchmark a due processi.
- Il Test ad otto processi su singola macchina ha un drastico calo di performance, evidentemente il piccolo processore ARM-v7 **Quad-Core** non riesce a supportate il carico di lavoro.
- I RaspBerry sono stati collegati via cavo ethernet con un bridge diretto, questo ha permesso una latenza media di **0.5ms**, nonostante ciò solo nella misurazione Cluster a otto processi sono state raggiunte tempistiche simili alla migliore misurazione ottenuta in singola macchina singola, ovvero a quattro processi.
- Possiamo anche notare che gli Speed-Up delle misurazioni Cluster hanno una buona scalabilità, questo portata a pensare che, in aggiunta di un altro dispositivo, avremmo potuto assistere ad una misurazione in **timing** migliore rispetto al Singolo RaspBerry.

Speed-Up & Timings:



Single RaspBerry - Timings				
Processi	100 (Gen)	1000 (Gen)	10000 (Gen)	100000 (Gen)
2	0,172146	1,6574	16,0323	160,425
4	0,062106	0,563923	5,62095	63,2047
8	2,2373	26,2302	209,002	~2000 (stimato)



RaspBerry Cluster - Timings				
Processi:	100 (Gen)	1000 (Gen)	10000 (Gen)	100000 (Gen)
2	0,459648	4,33535	43,3838	431,012
4	0,227314	2,16727	22,5768	216,393
8	0,190591	1,67895	16,7268	166,374